Report Group 27

Exercise 1

WS 2020 - 188.977 Grundlagen des Information Retrieval

Architecture

Tokenization

The tokenization of words is handled in the text2tokens-function which can be found in createindex.py. The following changes are applied to every word:

- lowercasing via string.casefold()
- stopword removal: every word appearing in the list of english stopwords from the nltk library is removed
- removal of special symbols via regular expression
- stemming: see chapter "Stemming" below

Inverted Index

The InvertedIndex-class is saved in invertedindex.py. An InvertedIndex-Object consists of two dictionaries:

- InvertedIndex.index represents the inverted index. The keys are strings representing the tokens (the tokenized words extracted from the articles). The values are objects of the class IndexNode, which represent the appearances of a token in the articles. More on the IndexNode-class below.
- InvertedIndex.doclengths is a simple dictionary which stores the token count of each article. The key is the ID of the article and the value is the token count.

Each appearance of a token in a word is represented by an instance of the IndexNode-class. An IndexNode object contains a dict with the article ID as key and the count of appearances in that article as value.

Because the native python data types we are using in our implementation are not memory efficient, roughly 8 GB of RAM are required in the process of creating the index of the evaluation set. Nevertheless, we chose standard python dictionaries for most of our implementations, because they offer search with an average time complexity of O(1). Furthermore, most modern machines can handle these requirements and it was explicitly stated that memory efficiency does not matter for this exercise.

Stemming

The PorterStemmer().stem(string) method from the nltk library is used for stemming.

Evaluation

Evaluation Results

Evaluation results for the evaluation on the **dev-set**.:

	MAP	NDCG@10	P@10	R@10
TFIDF	0.3728	0.6878	0.6635	0.1510
BM25	0.7148	0.9484	0.9192	0.2539

Evaluation results for the evaluation on the **evaluation-set**.:

	MAP	NDCG@10	P@10	R@10
TFIDF	0.0314	0.1570	0.1423	0.0311
BM25	0.2408	0.5959	0.5462	0.1382

As mentioned in the description of the Inverted Index, our program is not memory efficient as it takes up to 10GB of RAM. The run time of the index creation was around 1:40h - 2:00h on our machines (standard notebook PCs). It takes 1-2 minutes to load the evaluation index file at startup of the program, free text search in exploration mode then happens instantly. An evaluation run for the big dataset takes around 2 minutes.

Discussion of Results

TFDIF and BM25 are both related in how they calculate the relevance of a document given the search query. While simply ranking articles based on the number of occurrences of the word in relation to other articles is a reasonably good approximation in many cases our evaluation results show that there is quite a big headroom for improvement.

BM25 has some nice improvements over TFIDF, like normalizing the term frequency, which also has the effect that documents that match multiple search terms get better scores. It also accounts for document length, wherein our implementation if TFIDF is likely to favour long documents that contain the search term many times over short ones with fewer mentions.

Interestingly, we have a significantly better performance on the dev-set part of the dataset, we guess this is due to the dev-set not being a random sample, but rather chosen by intent, or another idea is that a small corpus is easier to search and therefore results in better matches.

Some ideas we had, though not implemented, is that occurrence in title or early in an article should be ranked higher, because a long article may drift off into discussion or related topics close to the end, while the intro part should usually be more on point and if we got a close match via a page title that indicates a good result.

How to run the Prototype

This description is for ubuntu, though it applies to most systems running Python3.

We are running **mongodb** as a document storage, for fast retrieval and as an extra to the exercise, since it is more practical than reading the documents via file offset. To clarify, we are not in any other way relying on mongodb in our indexing and search code, the database is only used to efficiently save and access the full text of the articles.

First we are going to install mongodb and virtualenv for our dependencies, then we install them from the requirements.txt file located in the /code directory:

```
apt-get update
apt-get install python3-venv  mongodb
python3 -m "venv" venv
pip install wheel (required in some distributions of python)
pip install -r requirements.txt
```

Our code expects mongodb to run on localhost with default port, if not `client = MongoClient()` needs to be modified accordingly.

If not already on the system, download the nltk stopwords in a plain python shell (# python3):

```
 >>> import nltk
 >>> nltk.download('stopwords')
```

After that everything should be set up to create the index, which is done via the createindex.py:

```
python createindex.py -f ../data/dev-set/ sampleindex
```

where "../data/dev-set/" is the location of the dataset and "sampleindex" will be the name of the created index file (which is placed in the "27_gir/data" folder).

Index creation can take multiple hours on the evaluation dataset (around two hours in our tests), therefore it is advised to run it in background and uninterrupted via the nohup command.

Once the index creation finished, we can interact with our dataset via the interact.py CLI interface:

```
python interact.py sampleindex
```

where "sampleindex" is the name of an index file in the "27_gir/data" folder.

It takes 1-2 minutes to load the index of the dev-set to memory.

Our menu then offers the following options:

```
### Main menu ###
Scoring: [TFIDF]
Choose a mode
 (X) Exploration mode
 (E) Evaluation mode
 (S) Switch to BM25
 (Q) Quit
>
```

Simply type the letter for the desired option and press enter to navigate through our application.

Optional Section

Feedback

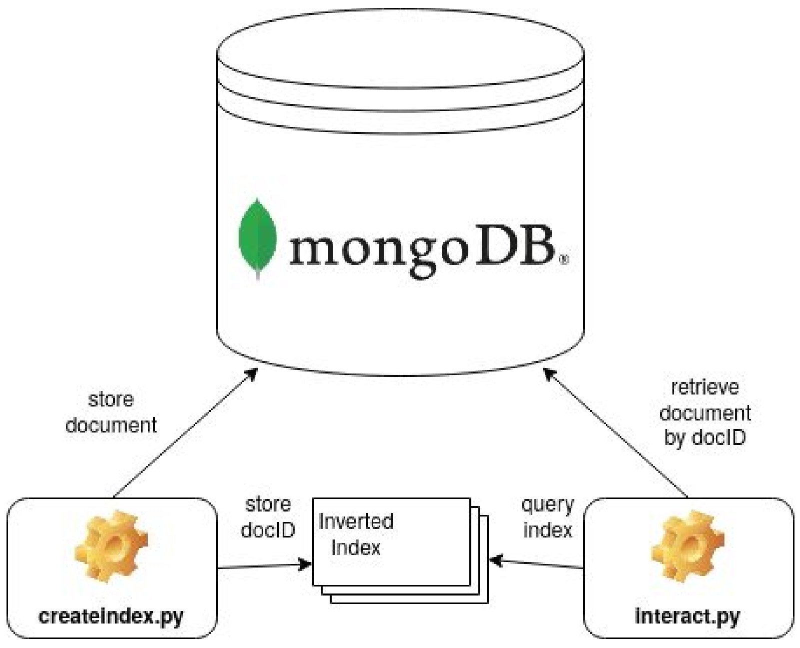
- The exercise was interesting and we appreciate the freedom of implementation
- The notes section could maybe be better structured, since I have to read through it multiple times looking for some hint specific to the problem I was currently trying to solve. Also there have been many questions answered in the TUWEL forum, which could be incorporated in the hints.
- We do not know if our implementation is correct and that is quite hard to reason about apart from looking at code and trial and error in exploration mode. To help students a quick suggestion would be to have an example query like “the quick brown fox jumps over the lazy dog” and list the top 10 suggestions plus scoring for the dev-set.

Interesting findings

An exceptionally good article at explaining TFIDF and BM25 is:
http://www.kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm25/

Approaches to obtain additional points

Instead of reading the files from disk we went for a more practical approach of using the real world document storage mongodb, which can also be run distributed allowing our architecture to scale:


mongoDB
store document
retrieve document by docID
store docID
Inverted Index
query index
createIndex.py
interact.py